

## Decision Trees

Many classes of problems can be formalized as **search problems in decision trees**.

We are going to look at two different applications of search trees:

First, we will solve the **n-queens problem** using backtracking search.

Second, we will discuss how you can write a program that beats you at your favorite **board game**.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

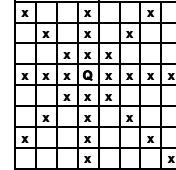
1

## Decision Trees

**Example I:** The n-queens problem

How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.



Here, the possible target squares of the queen Q are marked with an x.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

2

## Decision Trees

Let us consider the **4-queens** problem.

**Question:** How many possible configurations of 4×4 chessboards containing 4 queens are there?

**Answer:** There are  $16!/(12! \cdot 4!) = (13 \cdot 14 \cdot 15 \cdot 16)/(2 \cdot 3 \cdot 4) = 13 \cdot 7 \cdot 5 \cdot 4 = 1820$  possible configurations.

Shall we simply try them out one by one until we encounter a solution?

No, it is generally useful to think about a search problem more carefully and discover **constraints** on the problem's solutions.

Such constraints can dramatically speed up the search process.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

3

## Decision Trees

Obviously, in any solution of the n-queens problem, there must be **exactly one queen in each column** of the board.

Otherwise, the two queens in the same column could capture each other.

Therefore, we can describe the solution of this problem as a **sequence of n decisions**:

Decision 1: Place a queen in the first column.

Decision 2: Place a queen in the second column.

⋮

Decision n: Place a queen in the n-th column.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

4

## Backtracking in Decision Trees

There are problems that require us to perform an **exhaustive search** of all possible sequences of decisions in order to find the solution.

We can solve such problems by constructing the **complete decision tree** and then find a path from its root to a leaf that corresponds to a solution of the problem (breadth-first search often requires the construction of an almost complete decision tree).

In many cases, the efficiency of this procedure can be dramatically increased by a technique called **backtracking** (depth-first search).

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

5

## Backtracking in Decision Trees

**Idea:** Start at the root of the decision tree and move downwards, that is, **make a sequence of decisions**, until you either reach a solution or you enter a situation from where no solution can be reached by any further sequence of decisions.

In the latter case, **backtrack to the parent** of the current node and take a different path downwards from there. If all paths from this node have already been explored, backtrack to its parent.

Continue this procedure until you **find a solution** or establish that **no solution exists** (there are no more paths to try out).

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

6

## Backtracking in Decision Trees

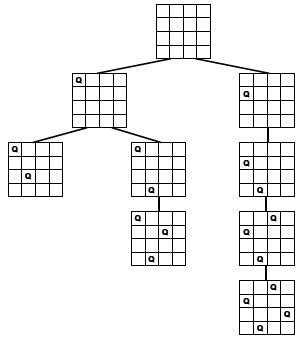
empty board

place 1<sup>st</sup> queen

place 2<sup>nd</sup> queen

place 3<sup>rd</sup> queen

place 4<sup>th</sup> queen



November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

7

## Two-Player Games with Complete Trees

**Example II:** Let us use similar search algorithms to write "intelligent" programs that **play games** against a human opponent.

Just consider this extremely simple (and not very exciting) game:

- At the beginning of the game, there are seven coins on a table.
- One move consists of removing 1, 2, or 3 coins.
- Player 1 makes the first move, then player 2, then player 1 again, and so on.
- The player who removes all remaining coins wins.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

8

## Two-Player Games with Complete Trees

Let us assume that the computer has the first move. Then, the game can be described as a **series of decisions**, where the first decision is made by the computer, the second one by the human, the third one by the computer, and so on, until all coins are gone.

The **computer** wants to make decisions that **guarantee its victory** (in this simple game).

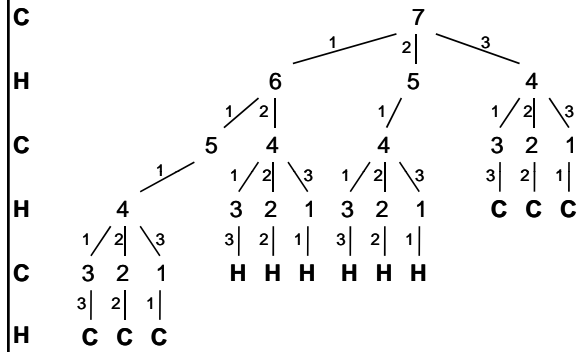
The underlying assumption is that the **human** always finds the **optimal move**.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

9

## Two-Player Games with Complete Trees



November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

10

## Two-Player Games with Complete Trees

The previous example showed how we can use a variant of backtracking to

- determine the computer's best move in a given situation and
- prune the search tree, that is, avoid unnecessary computation.

However, for more interesting games such as chess or go, it is **impossible** to check every possible sequence of moves.

The computer player then only looks ahead a certain number of moves and **estimates** the chance of winning after each possible sequence.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

11

## Two-Player Games

Therefore, we need to define a **static evaluation function**  $e(p)$  that tells the computer how favorable the current game position  $p$  is from its perspective.

In other words, the value of  $e(p)$  will be **positive** if a position is likely to result in a win for the computer, and **negative** if it predicts its defeat.

In any given situation, the computer will make a move that **guarantees** a maximum value for  $e(p)$  after a certain number of moves.

Again, the opponent is assumed to make **optimal moves**.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

12

## Two-Player Games

For example, let us consider **Tic-Tac-Toe** (although it would still be possible to search the complete game tree for this game).

What would be a suitable evaluation function for this game?

We could use the **number of lines** that are still open for the computer (X) minus the ones that are still open for its opponent (O).

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

13

## Two-Player Games


$$e(p) = 8 - 8 = 0$$

X		
O	X	

$$e(p) = 6 - 2 = 4$$

O	O	X
X	O	
X		

$$e(p) = 2 - 2 = 0$$

shows the weakness of this  $e(p)$

How about these?

O	O	X
	X	
X		

$$e(p) = \infty$$

X	X	
O	O	O
	X	

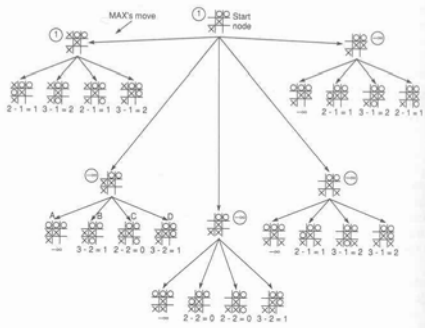
$$e(p) = -\infty$$

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

14

## Two-Player Games



November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

15

## Two-Player Games

In summary, we need **three functions** to make a computer play our favorite game:

- a function that generates all possible moves in a given situation,
- a static evaluation function  $e(p)$ , and
- a tree search algorithm.

This concept underlies most game-playing programs.

It is **very powerful**, as demonstrated, for example, by the program Deep Blue beating the world chess champion Gary Kasparov in 1997.

November 10, 2009

Introduction to Cognitive Science  
Lecture 17: Game-Playing Algorithms

16